45장. 프로미스

- 자바스크립트에서는 비동기 처리를 하기 위한 하나의 패턴으로 `콜백 함수`를 사용
- 콜백 패턴의 **단점**
  - 콜백 헬로 인해 `가독성 나쁨`
  - `에러 처리`가 곤란함
  - 여러 개의 `비동기 처리`를 한 번에 처리하는 데 한계가 있음
- 따라서 ES6에서는 비동기 처리를 위한 또 다른 패턴으로 **프로미스**를 도입했다.
- 프로미스는 콜백 패턴이 가진 단점을 보완하며 비동기 처리 시점을 명확하게 표현할 수 있다.

45.1 비동기 처리를 위한 콜백 패턴의 단점

45.1.1 콜백 헬

비동기 처리가 필요한 이유

```
// GET 요청을 위한 비동기 함수
const get = (url) => {
  const xhr = new XMLHttpRequest();
  xhr.open("GET", url);
  xhr.send();

  xhr.onload = () => {
    if (xhr.status === 200) {
        // 1. 서버의 응답을 반환
      return JSON.parse(xhr.response);
    }
    console.log(`${xhr.status}`);
  };
};

// 2. id가 1인 post 취득
const response = get("https://json~~");
console.log(response); // undefined
```

1. 비동기 함수 `get`이 호출되면 get 함수의 `실행 컨텍스트`가 생성되고 `콜 스택`에 푸시된다.
2. 이후 함수 코드 실행 과정에서 `xhr.load` 이벤트 핸들러 프로퍼티에 `이벤트 핸들러`가 `바인딩`된다.
3. `get` 함수가 종료되면 get 함수의 `실행 컨텍스트가 콜 스택에서 제거`된다.
4. 그후 곧바로 console.log(response)가 실행된다.
5. 서버로부터 응답이 도착하면 xhr 객체에서 `load 이벤트`가 발생한다.

> 이때 xhr.load 이벤트 핸들러는 load 이벤트가 발생하면 일단 태스크 큐에 저장되어 대기하다가, 콜 스택이 비면 이벤트 루프에 의해 콜 스택으로 푸시되어 실행된다. 따라서 xhr.load 이벤트 핸들러가 실행되는 시점에는 콜 스택이 빈 상태여야 하므로 console.log가 종료된 이후에 실행된다.
>
> 이처럼 비동기 함수는 비동기 처리 결과를 외부에 반환할 수 없고, 상위 스코프의 변수에 할당할 수도 없다.
>
> 따라서 비동기 함수의 처리 결과에 대한 후속 처리는 비동기 함수 내부에서 수행해야 한다.

- 이때 비동기 함수를 범용적으로 사용하기 위해 비동기 함수에 비동기 처리 결과에 대한 후속 처리를 수행하는 `콜백 함수`를 전달하는 것이 일반적이다.
- 팔요에 따라 `비동기 처리가 성공`하면 호출될 콜백 함수와 비동기 처리가 실패하면 호출될 콜백 함수를 전달할 수 있다.

```
// GET 요청을 위한 비동기 함수
const get = (url, successCallback, failureCallback) => {
  const xhr = new XMLHttpRequest();
  xhr.open("GET", url);
  xhr.send();

  xhr.onload = () => {
    if (xhr.status === 200) {
        // 서버 응답을 콜백 함수에 인수로 전달하며 호출하여 응답에 대한 후속 처리함.
      successCallback(JSON.parse(xhr.response));
    }else{
        // 에러 정보를 콜백 함수에 인수로 전달하며 호출하여 에러 처리함.
      failureCallback(xhr.status);
    }
  };
};

get("https://json~~", console.log, console.error); // 결과 잘 나옴
```

> 이처럼 콜백 함수를 통해 비동기 처리 결과에 대한 후속 처리를 수행하는 비동기 함수가 비동기 처리 결과를 가지고 또다시 비동기 함수를 호출해야 한다면 콜백 함수 호출이 중첩되어 복잡도가 높아지는 현상이 발생하는데, 이를 **콜백 헬**이라고 함.

```
// GET 요청을 위한 비동기 함수
const get = (url, callback) => {
  const xhr = new XMLHttpRequest();
  xhr.open("GET", url);
```

```
  xhr.send();

  xhr.onload = () => {
    if (xhr.status === 200) {
        // 서버 응답을 콜백 함수에 인수로 전달하며 호출하여 응답에 대한 후속 처리함.
      callback(JSON.parse(xhr.response));
    }else{
      console.error(`${xhr.status}`);
    }
  };
};

const url = "https://json~~";

// id가 1인 post의 userId 취득
get(`${url}/posts/1`, ({userId}) => {
    console.log(userId);
    // post의 userId를 사용해 user 정보 취득
    get(`${url}/posts/${userId}`, userInfo => {
        console.log(userInfo);
    })
});
```

- 위처럼 `GET` 요청을 통해 서버로부터 응답을 받고, 그 데이터를 사용해 또다시 GET 요청을 하다보면 가독성이 나빠져 실수를 유발하는 원인이 될 수 있다.

전형적인 콜백 헬 사례

```
get(`/step1`, a => {
    get(`/step2/${a}`, b => {
        get(`/step3/${b}`, c => {
            get(`/step4/${c}`, d => {
                console.log(d);
            })
        })
    })
});
```

- 4개의 비동기 요청이 중첩되어 있음.

45.1.2 에러 처리의 한계

- 비동기 처리를 위한 콜백 패턴의 문제점 중에서 가장 심각한 것은 **에러 처리**가 곤란하다는 것

```
try {
    setTimeout(() => {throw new Error('Error!');}, 1000);
```

```
}catch(e) {
    console.log('캐치한 에러', e); // 에러를 캐치하지 못한다.
}
```

45.2 프로미스의 생성

- `Promise` 생성자 함수는 비동기 처리를 수행할 콜백 함수를 인수로 전달받는데 이 콜백 함수는 `resolve` 와 `reject` 함수를 인수로 전달받는다.

```
// 프로미스 생성
const Promise = new Promise((resolve, reject) = > {
    // Promise 함수의 콜백 함수 내부에서 비동기 처리를 수행
    if(/* 비동기 처리 성공 */){
    resolve('result');
  } else{
    /* 비동기 처리 실패 */
      reject('failure reason');
    }
})
```

- Promise 생성자 함수가 인수로 전달받은 콜백 함수 내부에서 비동기 처리를 수행한다.
- 비동기 처리가 `성공` : **resolve** 함수 호출 / `실패` : **reject** 함수 호출

```
// GET 요청을 위한 비동기 함수
const get = url => {
  return new Promise((resolve, reject) = > {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();

    xhr.onload = () => {
        if (xhr.status === 200) {
        // 성공적으로 응답 전달받으면 resolve 함수 호출
            resolve(JSON.parse(xhr.response));
        }else{
        // 에러 처리를 위해 reject 함수 호출
            reject(new Error xhr.status);
        }
    };
};

get('https://~~');
```

- 프로미스는 다음과 같이 현재 비동기 처리가 어떻게 진행되고 있는지를 나타내는 `상태(state) 정보`를 갖는다.

프로미스의 상태 정보[2]	의미	상태 변경 조건
pending	비동기 처리가 아직 수행되지 않은 상태	프로미스가 생성된 직후 기본 상태
fulfilled	비동기 처리가 수행된 상태(성공)	resolve 함수 호출
rejected	비동기 처리가 수행된 상태(실패)	reject 함수 호출

- 생성된 직후의 프로미스는 기본적으로 `pending` 상태이고, 비동기 처리 결과에 따라 프로미스의 상태가 변경된다.
- `fulfilled` 또는 `rejected` 상태를 `settled` 상태라고 한다. (= 성공/실패 여부와 상관없이 비동기 처리가 수행된 상태)

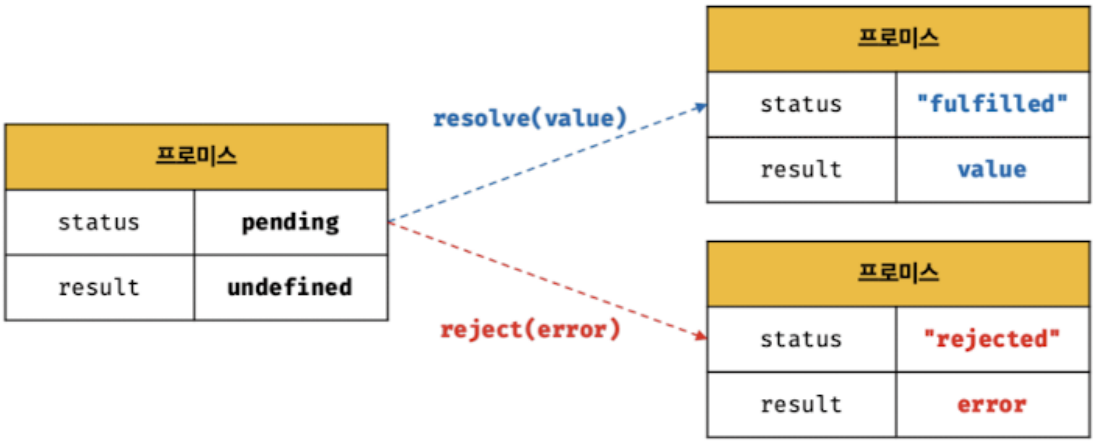

그림 45-1 프로미스의 상태

➡️ 즉, **프로미스는 비동기 처리 상태와 처리 결과를 관리하는 객체다.**

45.3 프로미스의 후속 처리 메서드

- 프로미스의 비동기 처리 상태가 변화하면 이에 따른 `후속 처리`를 해야 한다.

> 이를 위해 프로미스는 후속 메서드 then, catch, finally를 제공한다.

45.3.1 Promise.prototype.then

- then 메서드는 2개의 콜백 함수를 인수로 전달받는다.

1. fulfilled 상태일 때 호출되는 콜백 함수 → ⭕ 비동기 처리 성공 콜백 함수
2. rejected 상태일 때 호출되는 콜백 함수 → ❌ 비동기 처리 실패 콜백 함수

45.3.2 Promise.prototype.catch

- catch 메서드는 1개의 콜백 함수를 인수로 전달받는다.

- catch 메서드의 콜백 함수는 프로미스가 `rejected` 상태인 경우에만 호출된다.

```
new Promise(resolve => reject(new Error('rejected')))
    .catch(e => console.log(e));
```

45.3.3 Promise.prototype.finally

- finally 메서드는 1개의 콜백 함수를 인수로 전달받는다.
- finally 메서드의 콜백 함수는 프로미스의 성공 또는 실패와 상관없이 **무조건 1번 호출**된다.
- finally 메서드는 프로미스의 상태와 상관없이 공통적으로 수행해야 할 처리 내용이 있을 때 유용

```
new Promise(() => {})
    .finally(() => console.log('finally')); // finally
```

후속 처리 구현

```
const promiseGet = url => {
  return new Promise((resolve, reject) = > {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.send();

    xhr.onload = () => {
        if (xhr.status === 200) {
                // 성공
            resolve(JSON.parse(xhr.response));
        }else{
                // 실패
            reject(new Error xhr.status);
        }
    };
};

// promiseGet 함수는 프로미스를 반환
promiseGet('https://~~')
    .then(res => console.log(res))
    .catch(err => console.error(err))
    .finally(() => console.log('Bye'));
```

45.4 프로미스의 에러 처리

- 프로미스로 에러를 처리할 수 있는 방법 2가지

1. `then` 메서드의 두 번째 콜백 함수로 처리

2. `catch` 메서드를 사용해 처리
    - 2번 방법이 더 가독성 좋고 명확하기 때문에 에러 처리는 2번 방법을 권장

45.5 프로미스 체이닝

- `then`, `catch`, `finally` 후속 처리 메서드는 언제나 프로미스를 반환하므로 연속적으로 호출할 수 있는데, 이를 **프로미스 체이닝**이라 함.

> 프로미스는 프로미스 체이닝을 통해 비동기 처리 결과를 전달받아 후속 처리를 하므로 비동기 처리를 위한 콜백 패턴에서 발생하던 콜백 헬이 발생하지 않는다.
> 다만 프로미스도 콜백 패턴을 사용하므로 콜백 함수를 사용하지 않는 것은 아니다.

- 콜백 패턴은 가독성이 좋지 않은데 이 문제는 ES8에서 도입된 `async/await`를 통해 해결할 수 있다.

  async/await를 사용하면 프로미스의 후속 처리 메서드 없이 마치 동기 처리처럼 구현할 수 있다.

45.6 프로미스의 정적 메서드

45.6.1 Promise.resolve / Promise.reject

- 이미 존재하는 값을 래핑하여 프로미스를 생성하기 위해 사용
- Promise.resolve 메서드는 인수로 전달받은 값을 `resolve`하는 프로미스를 생성한다.
- Promise.reject 메서드는 인수로 전달받은 값을 `reject`하는 프로미스를 생성한다.

```
// 배열을 resolve/reject 하는 프로미스를 생성
const resolvedPromise = Promise.resolve([1,2,3]);
const rejectedPromise = Promise.reject(new Error('Error'));

// 위와 동일하게 동작
const resolvedPromise = new Promise(resolve => resolve([1,2,3]);
const rejectedPromise = new Promise((_,reject) => reject(new Error('Erro
r'));
```

45.6.2 Promise.all

- 여러 개의 비동기 처리를 모두 **병렬 처리**할 때 사용한다.

순차적으로 처리 ⇒ 약 6초 소요

```
const requestData1 = () => new Promise((resolve) => setTimeout(() => resolve(
const requestData2 = () => new Promise((resolve) => setTimeout(() => resolve(
const requestData3 = () => new Promise((resolve) => setTimeout(() => resolve(

// 세 개의 비동기 처리를 순차적으로 처리
const res = [];
```

```
requestData1()
  .then((data) => {
    res.push(data);
    return requestData2();
  })
  .then((data) => {
    res.push(data);
    return requestData3();
  })
  .then((data) => {
    res.push(data);
    console.log(res); // [1, 2, 3] ⇒ 약 6초 소요
  })
  .catch(console.error);
```

- 세 개의 비동기 처리는 서로 의존 X, 개별적으로 수행됨

  → 순차적으로 처리할 필요 X

병렬 처리 ⇒ 약 3초 소요

```
const requestData1 = () => new Promise(resolve => setTimeout(() => resolve
(1), 3000));
const requestData2 = () => new Promise(resolve => setTimeout(() => resolve
(2), 2000));
const requestData3 = () => new Promise(resolve => setTimeout(() => resolve
(3), 1000));

// 3개의 비동기 처리를 병렬로 처리
Promise.all([requestData1(), requestData2(), requestData3()])
    .then(console.log) // [1,2,3]
    .catch(console.error);
```

45.6.3 Promise.race

- Promise.all 메서드와 동일하게 프로미스를 요소로 갖는 배열 등의 이터러블을 인수로 전달받는다.
- Promise.all과의 **차이점**

  Promise.all 처럼 모든 프로미스가 fulfilled 상태가 되는 것을 기다리는 것이 아니라, 가장 먼저fulfilled 상태가 된 프로미스의 처리 결과를 `resolve` 하는 `새로운 프로미스` 를 반환한다.

```
const requestData1 = () => new Promise(resolve => setTimeout(() => resolve
(1), 3000));
const requestData2 = () => new Promise(resolve => setTimeout(() => resolve
(2), 2000));
const requestData3 = () => new Promise(resolve => setTimeout(() => resolve
(3), 1000));
```

```
Promise.race([requestData1(), requestData2(), requestData3()])
    .then(console.log) // 3
  .catch(console.error);
```

45.6.4 Promise.allSettled

- 전달받은 프로미스가 모두 `settled` 상태(비동기 처리 수행된 상태)가 되면 처리 결과를 **배열**로 반환한다.
- Promise.allSettled 메서드가 반환한 배열에는 fulfilled 또는 rejected 상태와는 상관없이

  Promise.allSettled 메서드가 인수로 전달받은 모든 프로미스들의 처리 결과가 모두 담겨 있다.

45.7 마이크로태스트 큐

```
setTimeout(() => console.log(1), 0);

Promise.resolve()
    .then(() => console.log(2))
    .then(() => console.log(3));
// 출력 순서 예상 : 1 - 2 - 3 ?
```

- 위 예제의 출력 순서는 2 -> 3 -> 1 이다.
  - 그 이유는 프로미스의 후속 처리 메서드의 콜백 함수는 태스크 큐가 아니라 `마이크로태스크 큐`에 저장되기 때문

> 콜백 함수나 이벤트 핸들러를 일시 저장한다는 점에서 태스크 큐와 동일하지만 **마이크로태스크 큐는 태스크 큐보다 우선순위가 높다.**

- 즉, 이벤트 루프는 콜 스택이 비면 먼저 마이크로태스크 큐에서 대기하고 있는 함수를 가져와 실행한다. 이후 마이크로태스크 큐가 비면 `태스크 큐`에서 대기하고 있는 함수를 가져와 실행한다.

45.8 fetch

- fetch 함수는 XMLHttpRequest 객체와 마찬가지로 HTTP 요청 전송 기능을 제공하는 `클라이언트 사이드 Web API`다.
- fetch 함수는 XMLHttpRequest 객체보다 사용법이 간단하고 프로미스를 지원하기 때문에 비동기 처리를 위한 콜백 패턴의 단점에서 자유롭다.
- fetch 함수에는 HTTP 요청을 전송할 URL과 HTTP 요청 메서드, HTTP 요청 헤더, 페이로드 등을 설정한 객체를 전달한다.

```
const promise = fetch(url, [, options]);
```

- **fetch 함수는 HTTP 응답을 나타내는 Response 객체를 래핑한 Promise 객체를 반환한다.**

    - fetch 함수가 반환하는 프로미스는 기본적으로 404나 500 같은 HTTP 에러가 발생해도 에러를 reject하지 않고 Response 객체를 resolve한다. 네트워크 장애나 CORS 에러에 의해 요청이 완료되지 못한 경우에만 프로미스를 reject한다.

- 따라서 fetch 함수를 사용할 때는 fetch 함수가 반환한 프로미스가 resolve한 불리언 타입의 ok 상태를 확인해 `명시적으로 에러를 처리` 해야 한다.

> 참고로 axios는 모든 HTTP 에러를 reject하는 프로미스를 반환한다. 따라서 모든 에러를 catch에서 처리할 수 있어 편리하다. 또한, axios는 인터셉터, 요청 설정 등 더 다양한 기능을 제공한다.

https://tlsdnjs12.tistory.com/26